Bull. Nov. Comp. Center, Comp. Science, 31 (2010), 1–16
© 2010 NCC Publisher

Introduction to the Atoment language*

I. S. Anureev

Abstract. The Atoment language is a domain-specific language of development for program verification methods. It is used in the multilanguage software system Spectrum of rapid development and testing of verification methods. The easy-to-use specialized language allows a user of the system to describe verification methods in a natural notation, verify algorithms for different object domains adding new languages for their representations as necessary, share verification methods with other users and combine them. In this paper, an introduction to the Atoment language including a description of its main entities and examples of their use are given.

Keywords: program specification, program verification, domain specific language, pattern matching, program model, information system

1. Introduction

A modern trend in the field of program verification is transition from the development of verification methods which are applied to small programs in model languages to development of verification methods which are applied to big software systems in industrial programming languages. The tendency is to pick out program properties which are important in practice and to develop specialized methods of analysis and verification for these cases. Unification and formalization of descriptions of these properties and verification methods for them are an important open problem. The use of combinations of verification methods is also peculiar to industrial verification. Development of tools for accumulation, analysis and formalization of experience in the field of integration of different verification methods is another important open problem.

The purpose of the project Spectrum is to develop a new approach to program verification which allows us to integrate, unify and combine verification methods, accumulate and use knowledge about them. A peculiarity of the approach is the use of the domain-specific language Atoment [1, 2] for development of program verification tools which allows us to represent both verification methods and data for them (program models, annotations, logical formulas) in an unified format. The system Spectrum [3] which is based on this language can be considered as both a specialized environment for development of program verification tools and an information system which

*Partially supported by RFBR under Grant 08-01-00899-a and Integration Project RAN 2/12.

accumulates knowledge in this field and provides access to it. In particular, program verification methods are knowledge represented in this system in the Atoment language. At present author's developments in the field [4, 5, 6, 7, 8, 9, 10] are integrated into the system.

The paper has the following structure. In Section 2 the requirements to a domain-specific language for development of program verification methods are formulated and it is shown what components of the Atoment language meet these requirements. The main semantic entities of the Atoment language and examples of their use are presented in Sections 3–7. Syntactic constructs of this language are described in Sections 8–9.

2. Requirements to the verification-oriented language

In this section we define the requirements that a domain-specific language for development of program verification methods should meet. A sufficiently large number of practical methods of program analysis and verification satisfy the following scheme of use:

1. Get a verification method, a program in a target programming language and an annotation that describes a program property to be analyzed.
2. Translate the annotated program into a graph form.
3. Apply a graph transformation that implements the method.
4. Return an interpretation of the final graph.

Static analysis methods, operational and axiomatic semantics methods, model checking methods, finite-state automaton methods, bisimulation methods and transformation semantics methods are well suited to this scheme.

Futher, a graph form of an annotated program or its (the graph form) textual representation in the verification-oriented language (VOL) will call a program model.

Translation of an annotated program into a program model must be one-to-one to provide the correctness of translation on the syntactic level and ease of handling of this program model. This requires a flexible syntax of VOL.

A more detailed analysis of transformations shows that the following tasks are often decided in these transformations: transformation combination tasks (for instance, sequential combination), local graph pattern matching, and local graph modification w.r.t. a pattern.

A local pattern matching on directed graphs is a task to check whether a node matches a pattern or not. The pattern depends on ancestors and descendants of the node of the specified depth. A local modification of a graph w.r.t. a pattern is a task to modify a node of the graph (and may be its descendants) in accordance with the pattern.

A modern programming language has a complex conceptual structure including hundreds of concepts. Therefore specification of a method in VOL should provide categorization of entities of the programming language, as well as creation and deletion of instances of these categories (concepts).

Let us summarize the requirements that VOL should meet. It should

1) specify programs, annotations and other data in a graph form;
2) specify transformation combination tasks;
3) specify a local graph pattern matching;
4) specify a local graph modification w.r.t. a pattern;
5) have flexible syntax for one-to-one representation of target programming languages;
6) categorize entities of target programming languages;
7) provide creation and deletion of instances of these categories.

The Atoment language meets all these requirements:

1) In this language, nodes of graphes are represented by atoments and labelled directed arcs are represented by labelled and ordinal properties (Section 3).
2) Transformation combination tasks are specified by contexts (Section 4), execution schemes (Section 6) and imperative actions of the standard library [2] of Atoment.
3) A local graph pattern matching is specified by search patterns (Section 5).
4) A local graph modification w.r.t. a pattern is specified by modification patterns (Section 5) and property declarations (Section 7).
5) Its flexible syntax is provided by different ways of building atoms from the Unicode symbols (Section 8) and a hierarchical way of definition of expressions (Section 9).
6) Categorization of entities of target programming languages is provided by logical properties (Section 3).
7) Creation and deletion of instances of these categories are provided by the logical property `[potential]` (Section 3).

3. Atoments and properties

The Atoment language is based on the concepts of an atom, element and property. The set of atoms and elements of the language forms the universe, the elements of which are called atoments. Atoments play the roles of data and actions. As an action, an atoment can be executed, it can change the state of the universe and return a value. As a data, an atoment can be used to represent a value returning an action and a data structure, the content of which is defined by a map from properties to atoments. A total function `st` from pairs `(P, A)`, where `P` is a property and `A` is an atoment, to atoments is called the state of the universe. The atoment `st(P, A)` is called the value of `P` w.r.t. `A` in `st`. A special atom `void` is a sign whether an atoment has a property. The atoment `A` has the property `P`, if `st(P, A)` $\neq$ `void`. The atoment `A` has not the property `P`, if `st(P, A) = void`. The atom `void` is also used as a value returned by an action, if this value is not important.

Properties are divided into labelled, logic and ordinal.

The set of labelled properties coincides with the set of sequences of the form `{`$\mathtt{A}_1$ `...` $\mathtt{A}_n$`}`, where $\mathtt{A}_i$ are atoments. In the case `{`$\mathtt{A}_1$`}`, where $\mathtt{A}_1$ is not an integer, brackets can be omitted. The value of the labelled property `P` of an atoment `A` is defined as `st(P, A)`.

The set of all logic properties coincides with the set of sequences of the form `[`$\mathtt{A}_1$ `...` $\mathtt{A}_n$`]`, where $\mathtt{A}_i$ are atoments.

Let the action `A.P ::= B` change the current state `st` to the state `st'`, where `st'(P, A) = B` for the pair `(P, A)` and `st'(V) = st(V)` for each other pair `V`. The action `A.P := B` is similar to the action `A.P ::= B` except for two cases:

- If `A` is an exception, then this action does nothing and returns `A`;
- If `A` is not an exception, and `B` is an exception, then this action does nothing and returns `B`.

Exceptions are special kinds of elements that are considered in this section below.

Example. Assume that `A` does not have the labelled property `color` and logic property `[traffic lights]`. The action `A.[traffic lights] := true` adds the property `[traffic lights]` to `A`. The action `A.color := green` adds the property `color` to `A` and assigns `green` to it. The action `A.color := red` changes the value of the property `color` from `green` to `red`. The action `A.color := void` deletes the property `color` from `A`.

The set of ordinal properties coincides with the set of integers. Integers are atoms. The value of the ordinal property `P` of an atoment `A` is defined as `st(P, A)`. The number of ordinal properties of an atoment is called its length. An atoment must satisfy the continuity property: if `i < j`, `A.i` $\neq$ `void`, `A.j` $\neq$ `void`, then `A.k` $\neq$ `void` for each `i < k < j`. An atoment with the ordinal properties from Min to Max is called a sequence from Min to Max. The labelled properties left and right define different kinds of sequences. A sequence A with the property left cannot have an ordinal property that is less than A.left. A sequence A with the property right cannot have an ordinal property that is greater than A.right. The action A.m := B for an ordinal property m depends on the kind of the sequence A.

Example. Let A have the ordinal properties 1, 2, 3 with their values 1, 4, 9. Let A.left = 0 and A.right = void. The action A.0 := 0 adds the property 0 to A and assigns 0 to it. The action A.-1 := 0 does nothing, since A.left = 0. The action A.6 := 36 adds the properties 4, 5, 6 to A and assigns null, null, 36 to them. A special atom null is used for filling holes in the sequence A to preserve the continuity property. The action A.3 := void does nothing, since deletion of the property 3 violates the continuity property. The action A.3 := 36 deletes the property 6 from A.

Let A, B be atoments, P be a property. Let A.P denote st(P, A). Let A = (P_1:B_1 ... P_n:B_n [Q_1] ... [Q_m]) mean that A has ordinal and labelled properties P_1, ..., P_n with values B_1, ..., B_n and logic properties [Q_1], ..., [Q_m] and has no other properties.

The atoment A is called a parent of B w.r.t. P, if A.P = B. The atoment A is called a parent of B, if A.P = B for some P. The atoment A is called a child of B w.r.t. P, if B.P = A. The atoment A is called a child of B, if B.P = A for some P. The atoment A is called an ancestor of B, if A.P = B or A.P is an ancestor of B for some P. The atoment A is called an ancestor of B w.r.t. P, if A.P = B or A.P is an ancestor of B w.r.t. P. The atoment A is called a descendant of B, if B is an ancestor of A. The atoment A is called a descendant of B w.r.t. P, if B is an ancestor of A w.r.t. P. The atoment A is called a forefather of B, if A is an ancestor of B and A has no parents. The atoment A is called a forefather of B w.r.t. P, if A is an ancestor of B w.r.t. P and A has no parent w.r.t. P.

Note. Atoments can be parents (ancestors) of each other.

Elements are divided into actual and potential. The element A is called potential if A has the property [potential]. The element A is called actual if A has no property [potential]. The set of actual elements is called the actual universe. Potential elements cannot have properties except [potential] and they cannot be executed. All actions over a state of the universe are executed in the actual universe.

Potential elements can be added to an actual universe, and actual elements can be deleted from it. Adding a potential element to the actual universe is called generation of an (actual) element. Generation of an element consists in a nondeterministic choice of a potential element and deletion of the property [potential] from it. Deletion of an element consists in deletion of all its properties and all properties of its parents that refer to it and addition of the property [potential] to it.

Let `new:(`A_1`:`B_1 `...` A_n`:`B_n`)` mean to generate an element `D` with the properties A_1, ..., A_n with the values B_1, ..., B_n and to return it, and `new:()` mean to generate an element without properties and to return it.

An action returns either a pure value, or `void`, or an exception. The atom `void` has a sign that the action does not return a pure value or an exception. Exceptions are special kinds of elements. The atoment `A` is called a pure value, if `A` $\neq$ `void` and `A` is not an exception.

An exception is an element with the obligatory property `exception` and the optional property `value`. The value of `exception` is the type of the returned exception or its description. The value of `value` is usually an action which returns this exception.

4. Contexts

Actions must be executed in execution contexts.

The execution context `ct` is an element with the obligatory property `[context]` and the optional property `value`. Values returning by actions which are executed in the context `ct` are stored in the property `value`. If the action `A` returns the value `V` in `ct`, then `ct.value := V`. The atoment `ct.value` is called a current value in `ct`. The context `ct` can have the property `parent`. The value of this property is a context. The context `ct.parent` is called a parent of `ct`. A context can inherit data and actions of its parent. A context can contain elements: `A` $\in$ `ct` iff `A.context = ct`.

Because the contexts are elements, they can be added to an actual universe (be generated) and be deleted from it. Generation of a context consists in generation of an element and addition of the property `[context]` to it. Deletion of a context consists in deletion of all elements belonging to it and this context itself.

The access to an atoment can be performed by a special property `uid`. The atom `A.uid` is called a unique identifier of `A`. If `A` and `B` have the property `uid` and `A` $\neq$ `B`, then `A.uid` $\neq$ `B.uid`.

Note. The property `uid` is not obligatory.

Variables are a special kind of elements which must belong to contexts. Variables are used to refer to atoments.

A variable is an element with obligatory properties `[variable]`, `context`, `value` and `name`. Let `A` be a variable. The element `A.context` is a context to which `A` belongs. The atoment `A.value` is called the value of `A`. The value of the variable `A` is an atoment to which `A` refers. The atoment `A.name` is called the name of `A`. The name of a variable is used for access to its value.

A context can have the property `[inheritvar]`. If `ct.[inheritvar]` $\neq$ `void`, `ct.parent` $\neq$ `void`, `x` is a variable of `ct.parent`, and a variable with the name `x.name` does not belong to `ct`, then one can refer to the variable `x` from `ct` by its name.

5. Search and modification patterns

Patterns are used to search for atoments which meet certain demands, generate and modify atoments. A sequence of atoms called pattern variables can be associated with a pattern. Patterns are divided into search patterns and modification patterns.

A search pattern defines a set of atoments which have certain properties with certain values. An atoment `E` matches the search pattern, if `E` belongs to this set. A search pattern is used to check whether an atoment matches it and to extract the values of the properties of this atoment and its descendants and store them in the pattern parameters in case of matching.

An atoment can have hidden properties which are not taken into account when this atoment matches a pattern. The property `[show]` is a sign whether matching takes into account the hidden properties. The hidden properties are taken into account iff the pattern has this property.

Example. The pattern `([figure])` defines atoments that have the property `[figure]`.

Example. The pattern `([figure] ~[triangle])` defines atoments that have the property `[figure]` and do not have the property `[triangle]`.

Example. The pattern `([figure] length:5)` defines atoments that have the logic property `[figure]` and the labelled property `length` with the value `5`.

Example. The pattern (`1:`a_1 `...` `n:`a_n) defines atoments that have the properties `1`, ..., `n` with the values a_1, ..., a_n.

The pattern (`1:`x_1 `...` `n:`x_n) with the parameters x_1, ..., x_n defines atoments that have the properties `1`, ..., `n`. The values of these properties are assigned to the parameters x_1, ..., x_n.

Example. The pattern `([figure] length:x ~width)` with the parameter `x` defines atoments that have the logic property `[figure]`, labelled property `length` and do not have the labelled property `width`. The value of the property `length` is assigned to the parameter `x`.

Example. The pattern `if:x then:y else:z` defines a set of `if` statements. Matching an `if` statement `E` this pattern assigns the values of `E.if`, `E.then`, `E.else` to the pattern parameters `x`, `y`, `z`.

Example. Let `E = (1:1 2:2 3:3 4:4 5:5)`. The pattern `(2#4:x) (-3#1:y) (7#10:z)` assigns a new element `new:(2:2 3:3 4:4)` to the parameter `x`, a new element `new:(1:1)` to the parameter `y` and a new element `new:()` with the empty set of properties to the parameter `z`.

A modification pattern is applied to an atoment and defines the following transformations:

- addition of properties to the atoment and its descendants;
- deletion of properties of the atoment and its descendants;

- change of values of properties of the atoment and its descendants.

A modification pattern is also used to generate elements with definite properties. In this case, new elements with an empty set of properties are generated and then they are modified w.r.t. the modification pattern.

Example. The action `modify:E match:([figure] length:5 ~width)` is equivalent to the action `E.[figure] := true; E.length := 5; E.width := void`.

Example. The action `modify:E match:(B:(C:3))` is equivalent to the action `E.B.C := 3`.

Example. The action `modify:E match:(B:(C:3) [new])` is equivalent to the action `x := new:(C:3); E.B := x`.

Example. The action `modify:E match:(inslab:A inslog:B insright:C inslabn:D)` adds the labelled properties of `A`, the logic properties of `B`, the ordinal properties of `C` and the labelled properties of `D` to `E`. The ordinal properties of `C` are added right from the ordinal properties of `E`. The values of the labelled properties of `A` change the values of the labelled properties of `E`. A labelled property of `D` is added to `E` only if `E` does not have this property. Let `E = (u:1 v:2 w:3 [c] 1:1 2:2)`, `A = (u:2 a:3)`, `B = ([b])`, `C = (1:3 2:4)` and `D = (v:5 r:6)`. Then `E′ = u:2 v:2 w:3 a:3 r:6 [c] [b] 1:1 2:2 3:3 4:4)`, where `E′` is the result of modification of `E`.

Example. The action `modify:E match:(del:lab del:log)` deletes all labelled and logic properties of `E`.

6. Execution schemes

Execution of atoments (actions) is described by execution schemes. An execution scheme defines a set of atoments and a way of execution of atoments from this set. An atoment satisfies this scheme if it belongs to this set.

Note. An atoment can satisfy several execution schemes.

An execution scheme is an element with the obligatory properties `ifpattern` and `context` and the optional properties `where`, `whereafter`, `var`, `id`, `[macro]` and `then` such that an element `new:(match:P var: S.ifpattern where:S.where)`, where `P` is the result of addition of the property `[!]` to `S.ifpattern`, is a conditional search pattern. The property `[!]` means that the matched element must have the properties defined by the pattern and no other properties. An execution scheme must belong to the context defined by the property `context`.

Let `S` be an execution scheme, `CSP` be a conditional search pattern for `S`.

The atoment `S.id` is called an identifier of `S`. It is used to refer to `S`. The properties `ifpattern`, `where`, `whereafter` and `var` are used to check whether an atoment satisfies `S`. The parameters of `CSP` specified by the property `var` are called the parameters of S. They are used to store the values of the properties of atoments which satisfy S.

The parameters of S are divided into executed and non-executed, and their values are divided into preliminary and final.

The sequence of the preliminary values of the parameters is the result of matching an atoment E and CSP. The final values of executed parameters are the results of execution of preliminary values of these parameters. The final values of non-executed parameters are the preliminary values of these parameters. The executed parameter is defined in `S.var` as the value of the property `exec`.

The properties `where` and `whereafter` define restrictions on the preliminary and final values of parameters of S, respectively.

Matching an atoment and a scheme in a context ct returns either `void` that means "matching is impossible", or the list of matched values of parameters of the scheme.

The properties `where`, `whereafter` and `var` can be absent. The absence of the properties `where` and `whereafter` means that there are no restrictions on the preliminary and final values of parameters of S, respectively. The absence of the property `var` means that S has no parameters.

Schemes are divided into defined and predefined. The defined schemes are schemes that have the property `then`. The properties `then` and `[macro]` define the way of execution of atoments that satisfy the scheme. The predefined schemes do not have the property `then`. The way of execution of atoments that satisfy these schemes is defined externally.

If an atoment is a name of a variable that is accessible in the current context, then execution of the atoment returns the value of this variable. Otherwise, execution of an atoment in a current context consists in a choice of an execution scheme that the action matches, execution of the body of the scheme for matched values of its parameters in a new local context which is a parent of the current context, and returning the result to the current context. If there is no scheme that the action matches, then execution returns this atoment itself. The property `[macro]` allows us to describe macros, mixing variables of the current and local contexts.

Example.

```
ifpattern:f(x) [macro] var:$x then:(z := 5; eval:x);
z := 0;
f(z:=z+1)
```

Here `x` is the executed parameter of the pattern `ifpattern:f(x) var:$x`, since `x` has the prefix `$`. The action `f(z:=z+1)` in the context `ct` is equivalent to the action `z := 5; z:=z+1` in a new context `ct`$'$, and the action `z:=5` being executed in `ct`$'$ and the action `z:=z+1` being executed in `ct`. The action `eval:x` from the standard library [2] of the Atoment language executes an atoment that is the value of the action `x`.

7. Property declarations

There is a special kind of properties called functional properties. Functional properties are defined by property declarations.

Property declarations are used to redefine the base operations for a concrete property: assign a value to the property, get the value of the property, delete the property, check whether an atoment has the property. In addition, a property declaration defines a set of atoments, to which the redefined operations are applicable.

An atoment `A` is called a property declaration, if it has the obligatory property `prop` and optional properties `var`, `set`, `get`, `where` and `[readonly]`. The atoment `A.prop` defines a property `P` for which the base operations are specialized. The atoment `A.var` defines the parameters for the base operations and has the form `(A B)`. The atoments `A.set` and `A.get` define the base operations: assign a value `B` to the property `P` of the form `A.P := B` and get the value of the property `P` of the form `A.P`. The atoment `A.where` defines a set of atoments, to which the base operations specified by the properties `set` and `get` are applicable. The logic property `[readonly]` means that the value of the property `P` does not change.

The base operation of deletion of the property `P` is expressed by `A.P := void`, and the base operation of checking whether the atoment `A` has the property `P` is expressed by `A.P` $\neq$`void`.

Example. Let us define the property stack in the following way:

```
(prop:stack var:(A B) where:(A.[stack])
 set:
 (if:(B = void)
  then:
  (if:(A.max != void)
   then:(x := A.max; A.max := x - 1; A.x := void))
  else:
  (if:(A.max = void)
   then:(A.max := 0; A.0 := B)
   else:(x := A.max + 1; A.max := x; A.x := B))
 get:(if:(A.max != void) then:(x := A.max; return(A.x))
```

This property implements a stack on the sequence from `0` to $+\infty$. The property `max` defines the number of the up element of the stack, and the ordinal properties `0`, ..., `max` define the elements of the stack. An example of creation, use and deletion of a stack is described below.

```
A.[stack] := true; // A = ([stack])
A.stack := 0; // A = ([stack] max:0 0:0)
A.stack; // returns 0
A.stack := 1; // A = ([stack] max:1 0:0 1:1)
A.stack; // returns 1
A.stack := void; // A = ([stack] max:0 0:0)
A.stack; // returns 0
A.stack := void; // A = ([stack])
A.stack; // returns void
A.stack := void; // A = ([stack])
A.stack; // returns void
A.[stack] := void; // A = ()
A.stack // returns void
```

8. Atoms and properties

Constructs of the Atoment language are built from the Unicode symbols. Three groups of Unicode symbols play a special role. These are white spaces, special symbols and brackets. A white space is defined as any character with the Unicode class Zs (which includes the space character), as well as the horizontal tab character, the vertical tab character, and the form feed character. The set of special symbols includes

```
" ~ ‘ @ # $ % ^ & * - + : ; ′ . < > / , | \ = ? !
```

The set of brackets includes

```
( ) { } [ ]
```

Constructs of the language include atoms, properties and expressions.

A sequence of the Unicode symbols is called an atom, if it has one of the forms:

1) a sequence of the Unicode symbols, where each occurence of a bracket, a white space, or a special symbol precedes the symbol `\`;
2) the symbol `:` or a sequence of special symbols, except `"` and `\`, that is not ended by the symbol `:`;
3) `A"B"CB"`, where `A` is a (maybe empty) sequence of the Unicode symbols of the form (1), `B` is a nonempty sequence of the Unicode symbols that does not contain the symbol `"`, `C` is a sequence of the Unicode symbols that does not contain `B"` as a subsequence;
4) `A""C"`, where `A` is a (maybe empty) sequence of the Unicode symbols of the form (1), `C` is a sequence of the Unicode symbols, where each occurence of the symbol `"` precedes the symbol `\`;

5) an integer;
6) a real number;
7) `{`A_1 `...` A_n`}` or `'{`A_1 `...` A_n`}`, where A_i are atoms of the form (1)-(6);
8) `[`A_1 `...` A_n`]` or `'[`A_1 `...` A_n`]`, where A_i are atoms of the form (1)-(6);
9) `'A`, where `A` is an atom of the form (1)-(4).

Example. Let us consider different kinds of atoms used to define lexical constructs of the C language. Atoms of the form `c"'"B"` or `cL"'"B"`, (here `B` is a sequence of special symbols and representable symbols of the C language except `'`, `\` and new line) represent symbol constants of the C language.

Atoms of the form `s""B"` or `sL"B"`, where `B` is a sequence of special symbols and representable symbols of the C language except `"`, `\` and new line, represent string constants of the C language.

Atoms of the form `com"d"Bd"`, where `B` is any sequence of Unicode symbols that does not contain the subsequence `d"`, represent comments of the C language.

Atoms have predefined properties that can be used in the search and modification patterns and the operations `.`, `::=` and `:=`. The set of predefined properties depends on the kind of atoms.

Atoms of all kinds have the logic property `[atom]`.

Let `E` be an atom of the form (1) represented by a sequence of the length `n`. Then `E` has the logic property `[pure atom]`, the labelled property `length` with the value `n` and the ordinal properties `1`, ..., `n`, the values of which are the corresponding symbols of this sequence.

Example. The atom `if` matches the pattern `([pure atom] 1:i 2:f)`. The action `modify:if match:(1:o)` replaces `if` by `of`.

Let `E` be an atom of the form (2) represented by a sequence of length `n`. Then `E` has the logic property `[special atom]`, the labelled property `length` with the value `n` and the ordinal properties `1`, ..., `n`, the values of which are the corresponding symbols of this sequence.

Let `E` be an atom of the form (3). Then `E` has the logic property `[text atom]`, the labelled properties `type`, `delimiter` and `text` with the values `A`, B' and C', where B' and C' are atoms of the form (1) that are the result of replacement of each occurence of a bracket, a white space, or a special symbol `S` in `B` and `C` by `\S`.

Example. The action `new:([text atom] delimiter:\' text:A)` generates the atom `"'"A'"`.

Example. The action

```
new:([text atom] type:MultilineComment delimiter:**
text:This\ is\ a\ multiline\ comment\,
that\ does\ not\ contain\ the\ sequence\ of\ symbols\ **.)
```

generates the atom

```
MultilineComment"**"This is a multiline comment
that does not contain the sequence of symbols **".
```

Let `E` be an atom of the form (4). Then `E` has the logic property `[string atom]`, the labelled properties `type` and `text` with the values `A` and `C'`, where `C'` is an atom of the form (1) that is the result of replacement of each occurence of a bracket, a white space, or a special symbol `S` in `C` by `\S`.

Example. The action `new:([string atom] text:green)` generates the atom `""green"`.

Example. The action `new:([string atom] type:color text:green)` generates the atom `color""green"`.

Let `E` be a sequence of the form (5) represented by a sequence of digits of length `n` possibly with a sign. Then `E` has the obligatory logic property `[integer]`, the optional logic property `[-]` meaning that `E` has a sign, the labelled property `length` with the value `n` and the ordinal properties `1`, ..., `n`, the values of which are the corresponding digits in the representation of `E`.

Let `E` be an atom of the form (6). Then `E` has the obligatory logic property `[real]`, the optional logic property `[-]` meaning that `E` has a sign, and the labelled properties `int`, `frac` and `order`, the values of which are atoms of the form (5) representing integer, fractional and order parts of the real number.

Example. The atom `10.12E-15` matches the pattern `({~[-] int:10 frac:12 order:-15})`.

Let `E` be an atom of the form (7) represented by a sequence of length `n`. Then `E` has the logic property `[labelled property]`, the labelled property `length` with the value `n` and the ordinal properties `1`, ..., `n`, the values of which are the corresponding atoms of the form (1)-(6).

Let `E` be an atom of the form (8) represented by a sequence of length `n`. Then `E` has the logic property `[logic property]`, the labelled property `length` with the value `n` and the ordinal properties `1`, ..., `n`, the values of which are the corresponding atoms of the form (1)-(6).

An atom of the form `'A` has the same properties that the atom `A` has and has an additional logic property `[hidden]`.

Atoms of the forms (1)-(4) and (6) have the additional logic property `[labelled property]`. Atoms of the form (5) has the additional logic property `[ordinal property]`.

Thus properties are atoms of a special kind. Hidden properties have one of the forms: (9), `'{`A_1 `...` A_n`}` or `'[`A_1 `...` A_n`]`. Thus only labelled and logic properties can be hidden.

9. Expressions

Expressions represent atoments. For each kind of expressions there is an algorithm of translating them to atoments. According to the translation way, expressions are divided into expressions of order `0`, expresions of order `1`, and so on. Expressions of order `0` are directly traslated to atoments. Expressions of order `k`, where `k>0`, are translated to expressions of orders less than `k`.

Note. Algorithms of translation can be nondeterministic.

An expression of order `0` has one of the forms:

1) an atom or `()`;
2) a logic property;
3) `N:B`, where `N` is a labelled or ordinal property, `B` is an expression of the form `1` or `4`;
4) `(A`$_1$ `...` `A`$_n$`)`, where `A`$_i$ are expressions of the form `2` or `3`.

Let us consider translation of expressions of order `0`.

Example. The expression `(1:red 2:yellow 3:green)` defines an element with the ordinal properties `1`, `2` and `3` with the values `red`, `yellow` and `green`.

Example. The expression `([traffic light] colour:green)` defines an element with the logic property `[traffic light]` and the labelled property `colour` with the value `green`.

Example. The expression `(1:a 2:(3:b) 3:c)` defines an element with the ordinal properties `1`, `2` and `3` with the values `a`, E and `c`, where D is an element with the property `3` with the value `b`.

Expressions of higher orders provide more flexible and easy-to-use syntax in comparison with expressions of order `0`. Let us consider the main kinds of these expressions.

List expressions have one of the forms:

1. an atom or `()`;
2. a logic property;
3. `N:B`, where `N` is a labelled or ordinal property, `B` is a list expression of the form `1` or `4`;
4. `(A`$_1$ `...` `A`$_n$`)`, where `A`$_i$ are list expressions.

They allow us to use the list notation, omitting ordinal properties.

Example. The list expression `(red yellow green)` is reduced to the expression `(1:red 2:yellow 3:green)`.

Prefix expressions `F(X`$_1$`, ..., X`$_n$`)` are translated to expressions `(applyfun:F 1:X`$_1$ `...` `n:X`$_n$`)`. They allow us to write expressions in the prefix form.

Example. The prefix expression `sin(x)` is reduced to the expression `(applyfun:sin 1:x)`.

The ambiguity between one-place prefix expressions and list expressions is resolved depending on the presence of a white space before the bracket:

- `(a (b))` is translated to `(1:a (1:b))`;
- `(a(b))` is translated to `(1:(applyfun:a 1:b))`.

Infix expressions (`A`$_1$ `op ... op A`$_n$) are translated to (`applyfun:x 1:A`$_1$ `... n:A`$_n$). Infix expressions allow us to write binary operations in the infix form and use associativity of operations. Binary operations are divided into predefined (in the standard library of the Atoment language) and defined (by execution schemes). The execution scheme for the binary operation `op` has the pattern `A op B` with parameters `A` and `B`. Associativity of the operation is defined by the property `[assoc]` of this scheme.

Binary operations have the priority that defines the order of their application.

Example. The expression `(x+y*z)` is translated to the expression `(applyfun:+ 1:x 2:(applyfun:* 1:y 2:z))`, since the priority of `*` is greater than the priority of `+`.

Other kinds of expressions are defined in the standard library [2] of the Atoment language.

Execution of an expression consists of translation of the expression to an atoment and execution of this atoment. The auxiliary context is used to store the results of traslation. This context is deleted, when execution of the expression is terminated.

10. Conclusion

In this paper, we formulate the requirements that a domain-specific language for development of program verification methods should meet and present the verification-oriented language Atoment that meet all these requirements. The main entities of this language are described and examples of their use are given. We plan to use the Atoment language in the verification system Spectrum for development of program models of programming languages that are extensively used in practice (Java, C/C++, C# and so on), formal executable specifications of these languages on the basis of operational ontological approach [11], and the methods of specification and verification of program models and software systems.

References

[1] Anureev I.S. The Atoment Language: Syntax and Semantics. – Novosibirsk, 2010. – 47 p. – (Rep./RAS. Sib. branch. IIS; N 157) (in Russian).

[2] Anureev I.S. The Atoment Language: Standard Library. – Novosibirsk, 2010. – 47 p. – (Rep./RAS. Sib. branch. IIS; N 158) (in Russian).

[3] Nepomniashy V.A., Anureev I.S., Atuchin M.M., Maryasov I.V., Petrov A.A., Promsky A.V. C program verification in the multilanguage system Spectrum // Modelling and Analysis of Information Systems. – 2010. – N 4. – (to appear, in Russian).

[4] Shilov N.V., Anureev I.S., Bodin E.V. Generation of correctness conditions for imperative programs // Programming and Computer Software. – 2008. – N 6. – P. 307–321.

[5] Nepomniaschy V. A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs: C-Light language and its formal semantics // Programming and Computer Software. – 2002. – N 6. – P. 314–323.

[6] Nepomniaschy V. A., Anureev I.S., Promsky A.V. Towards verification of C programs: axiomatic semantics of the C-kernel Language // Programming and Computer Software. – 2003. – N 6. – P. 338–350.

[7] Nepomniaschy V. A., Anureev I.S., Dubranovsky I.V., Promsky A.V. Towards verification of C# programs: A three-level approach // Programming and Computer Software. – 2006. – N 4. – P. 190–202.

[8] Anureev I.S. A three-stage method of C program verification // Joint NCC&IIS Bulletin. Ser. Comput. Sci. – 2008. – Vol. 28 – P. 1–30.

[9] Anureev I.S., Maryasov I.V., Nepomnyaschy V.A. C-programs verification on basis of mixed axiomatic semantics // Modelling and analysis of information systems. – 2010. – N 3. – P. 1–23 (in Russian).

[10] Anureev I.S. Data structure elimination method based on formula rewriting systems // Programming and Computer Software. – 1999. – N 4. – P. 5–15.

[11] Anureev I.S. Operational ontological approach to formal programming language specification // Programming and Computer Software. – 2009. – N 1. – P. 35–42.